

# Solving the TTC 2011 Compiler Optimization Case with GrGen.NET

Sebastian Buchwald      Edgar Jakumeit

Karlsruhe Institute of Technology (KIT)

buchwald@kit.edu

The challenge of the Compiler Optimization Case [2] is to perform local optimizations and instruction selection on the graph-based intermediate representation of a compiler. The case is designed to compare participating tools regarding their performance. We tackle this task employing the general purpose graph rewrite system GRGEN.NET ([www.grgen.net](http://www.grgen.net)).

## 1 What is GrGen.NET?

GRGEN.NET is an application domain neutral graph rewrite system, which started as a helper tool for compiler construction. It is well suited for solving this challenge due to its origin, thus our solution can be seen as a reference, even though the origins were left behind a while ago. The feature highlights regarding practical relevance are:

**Fully Featured Meta Model:** GRGEN.NET uses attributed and typed multigraphs with multiple inheritance on node/edge types. Attributes may be typed with one of several basic types, user defined enums, or generic set, map, and array types.

**Expressive Rules, Fast Execution:** The expressive and easy to learn rule specification language allows straightforward formulation of even complex problems, with an optimized implementation yielding high execution speed at modest memory consumption.

**Programmed Rule Application:** GRGEN.NET supports a high-level rule application control language, Graph Rewrite Sequences (GRS), offering sequential, logical, iterative and recursive control plus variables and storages for the communication of processing locations between rules.

**Graphical Debugging:** GRShell, GRGEN.NET's command line shell, offers interactive execution of rules, visualising together with yComp the current graph and the rewrite process. This way you can see what the graph looks like at a given step of a complex transformation and develop the next step accordingly. Or you can debug your rules and sequences.

**Extensive User Manual:** The GRGEN.NET User Manual [1] guides you through the various features of GRGEN.NET, including a step-by-step example for a quick start.

## 2 Constant Folding

The first task is to perform constant folding. Constant folding transforms operations which have only `Const` operands into a `Const` itself, e.g. to transform  $1 + 2$  into  $3$ . This is a local optimization requiring only rules referencing local graph context.

## 2.1 Driver and Data Flow

Constant folding is carried out from the driver sequence given in Figure 1, which performs inside a main loop in each step i) first constant folding along data flow with a wavefront algorithm, and then ii) control flow folding together with clean-up tasks.

Graph rewrite sequences are the rule application control language of GRGEN. The most fundamental operation is a rule application denoted by the rule name, with parameters given in parenthesis, and optionally an assignment of output values to variables (given in parenthesis, too). By using all bracketing `[r]` we execute the rule `r` for all matches in the host graph. The then-right operator `>` executes the left sequence and then the right sequence, returning as result of execution the result of the execution of the right sequence. The potential results of sequence execution are *success* equaling `true` and *failure* equaling `false`; a rule which matches (at least once) counts as success. With the postfix star `*` we iterate the preceding sequence as long as it succeeds. The result of a star iteration is always success, in contrast to the plus `+` postfix which requires the preceding sequence to match at least once in order to succeed; so a sequence of rules with plus postfixes linked by strict disjunction operators `|` succeeds (`true`) if one of the rules matches. Conjunction `&` is available as well, so are the lazy versions `&&` and `||` of the operators not executing the right sequence in case the result of the left sequence already determines the outcome. The prefix exclamation mark operator negates the result of sequence execution, in Figure 1 we negate the value from variable `isEmpty`. The backslashes `\` allow to concatenate multiple lines into one.

```
xgrs now:set<FirmNode>=set<FirmNode>{} ;> next:set<FirmNode>=set<FirmNode>{} ;>\
  ( [collectConstUsers(dummy, now)] ;>\
    (wavefront(now, next) ;> now.clear() ;> tmp=now ;> now=next\
      ;> next=tmp ;> isEmpty=now.empty() ;> !isEmpty)* ;>\
    [foldAssociativeAndCommutative] ;>\
    (foldCond+ | removeUnreachablePhiOperand+ |\
      removeUnreachableBlock+ | removeUnreachableNode+)* ;>\
    fixEdgePosition* ;>\
    (mergeConsts+ | removeUnusedNode+ | mergeBlocks+)\
  )*
```

Figure 1: Driver sequence for constant folding.

In the driver sequence two empty storage sets are created initially for the wavefront algorithm: `now` which will contain the users of constant nodes which are visited in this step and `next` which will contain the users of constant nodes which will be visited in the next step. Then the main loop is entered. There the rule `collectConstUsers` is executed on all available matches, collecting all users of `Const` nodes available in the graph into the storage set `now` (the `dummy` variable was never assigned and the `collect-ConstUsers` rule was specified to search for a valid entity if the first parameter is undefined). This is the initial stone thrown into the water of our program graph. From it on a wavefront is iterated following the data flow edges until it comes to a halt because no new constants were created anymore. A wavefront step is encapsulated in the subsequence `wavefront` given in Figure 2; it visits all operations of its first argument, and adds the operations which are users of the constants it created newly with constant folding into the set given as its second argument. After this subsequence was called on the `now` argument the `now` and `next` sets are swapped (the set referenced by `now` is cleared, `now` is assigned the set referenced by the `next`, and `next` is assigned the empty set previously pointed to by `now`). The wavefront iteration is stopped when the `now` set of the next iteration step is empty.

A wavefront step defined by the subsequence given in Figure 2 iterates with a `for` loop over the users of

```

def wavefront(constUsersNow:set<FirmNode>,\
              constUsersNext:set<FirmNode>) {\
  for{\
    cu:FirmNode in constUsersNow;\
    ((c)=foldNot(cu) || (c)=foldBinary(cu) ||\
      (c)=foldPhi(cu) & false\
    )\
    && [collectConstUsers(c, constUsersNext)]\
  } ;> constUsersNow.clear()\
}

```

Figure 2: Wavefront step for constant folding.

constants contained in the storage set parameter `constUsersNow`, binding them to an iteration variable `cu` of type `FirmNode`. This constant is used as input to the rules `foldNot`, `foldBinary` and `foldPhi` really doing the folding in case all of the arguments to the operations they handle are constant. They return the newly created constant by folding, the rule `collectConstUsers` then adds all users of this constant to the `constUsersNext` storage set.

Before we take a closer look at the rules, let us start with a short introduction into the syntax of the rule language: Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The pattern part is built up of node and edge declarations or references with an intuitive syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type. An edge `e` with source `x` and target `y` is declared by `x -e:t-> y`, whereas `-->` introduces an anonymous edge of type `Edge`. Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively. Attribute conditions can be given within `if`-clauses.

The rewrite part is specified by a `replace` or `modify` block nested within the rule. With `replace`-mode, graph elements which are referenced within the `replace`-block are kept, graph elements declared in the `replace`-block are created, and graph elements declared in the pattern, not referenced in the `replace`-part are deleted. With `modify`-mode, all graph elements are kept, unless they are specified to be deleted within a `delete()`-statement. Attribute recalculations can be given within an `eval`-statement. In addition to rewriting, GRGEN supports relabeling, we prefer to call it retyping though. Retyping is specified with the syntax `y:t<x>`: this defines `y` to be a retyped version of the original node `x`, retyped to the new type `t`; for edges the syntax is `-y:t<x>->`.

These and the language elements we introduce later on are described in more detail in our solution of the Hello World case [3], and especially in the extensive GrGen.NET user manual [1].

Figure 4 shows the constant folding rule for Binary operations. It takes the node with the `Const` operand as parameter and returns the folded `Const`. The rule matches both `Const` operands and has an `alternative` statement that contains one case for each binary operation. Within the cases the value of the new `Const` is computed. In almost all cases the computation is as simple as in the `foldAdd` case. The only exception is the `foldCmp` case that also needs to consider the `relation` of the `Cmp` to compute the resulting value of the `Const`. Finally, at the end of the rule execution, the `exec` statement relinks the users of the `Binary` to the newly created `Const` and deletes the `Binary` from the graph (by executing the given sequence). The constant folding for unary `Not` nodes is straight forward and not further discussed;  $n$ -ary `Phi` nodes are of interest again. Figure 5 shows the corresponding rule. It first matches the `Phi` node and one `Const` operand and then iteratively edges to the same operand and to the `Phi` itself. If this covers all edges we can replace the `Phi` with the `Const` operand.

## 2.2 Control Flow and Cleanup

Let us mentally return to the driver sequence in Figure 1; after the wavefront which folded alongside data flow has collapsed, execution continues with folding condition nodes at the interface of data flow to control flow and with folding control flow proper (jumps and blocks). In addition there are some clean up task left to be executed. If no control flow was folded leading to further data flow folding possibilities, the main loop is left.

The rule shown in Figure 6 is responsible for folding Conditional jumps. Depending on the value of the constant, either the edge of type `True` gets deleted and the edge of type `False` retyped to an edge of type `Controlflow`, or the edge of type `False` gets deleted and the edge of type `True` retyped to an edge of type `Controlflow`. Since there is only one jump target left, we also retype the conditional jump to a simple jump of type `Jump`. Due to the folding of condition jumps, there may be unreachable blocks which can be deleted. We use two rules to remove unreachable code: the rule shown in Figure 7 deletes unreachable Blocks, the rule shown in Figure 8 deletes nodes without a Block. Furthermore, we also need to adapt Phi nodes if they have an operand without a `Controlflow` counterpart in the Block. Figure 9 shows the corresponding rule that matches a Phi and deletes an operand that has no `Controlflow` counterpart. After the deletion we fix the position of all edges using the `fixEdgePosition` rule.

The solution contains two rules `removeUnusedNode` and `mergeBlocks` that simplify the graph. The first rule removes a node that is not used, i.e. a node which has no incoming edges. This rule is similar to the `removeUnreachableBlock` rule. If there is a Block `block1` with only one outgoing edge of type `Controlflow` and this edge leads to a node of type `Jump` contained in Block `block2`, then we remove the `Jump` and merge `block1` and `block2` with the second rule. This rule is able to remove the chain of “empty” Blocks that occurs in the running example of the case description.

The verifier mentioned in the case description was implemented with the tests in `Verifier.gri`, called from a subsequence `verify` defined in `Verifier.grsi`; the subsequence is used with the statement `validate xgrs verify` before and after the transformations checking the integrity of the graph.

## 2.3 Folding More Constants

Figure 10 shows a rule that folds constants for the term  $(x * c1) * c2$  where  $*$  is a associative and commutative operation. This rule enables us to solve the test cases provided by the GReTL solution. In contrast to the GReTL rule, this rule does not change the structure of the graph.

## 3 Instruction Selection

The instruction selection task transforms the intermediate representation (IR) into a target-dependent representation (TR). It can be considered as some kind of model transformation. The TR supports immediate instructions, i.e. a `Const` operand can be encoded within the instruction. Our solution consists of one rule per target instruction which means we have at most two rules for each IR operation: one for the immediate variant and one for the variant without immediate.

Before we start matching all immediate operations, we apply the auxiliary rule shown in Figure 11. It matches commutative operations with a `Const` operand at position 0, but not at position 1, and then exchanges these operands to ensure that the constant operand is at position 1. Thus, the rule ensures deterministic behaviour in case of two constant operands.

Figure 12 shows exemplarily the rule that creates an `AddI` operation. It retypes the `Add` to an `TargetAddI`, deletes the `Dataflow` edge to the constant and stores the value of the `Const` node in the value

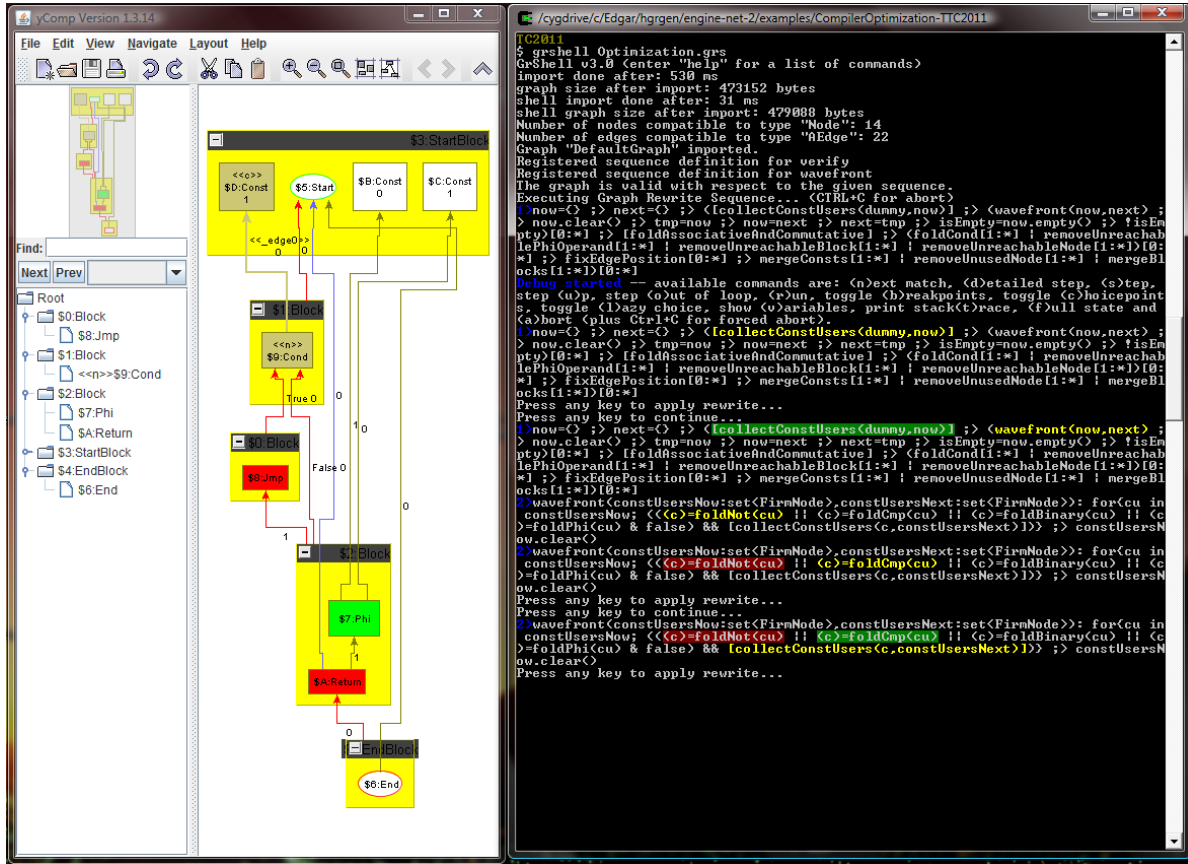


Figure 3: A situation from constant folding

attribute. Since the other rules for immediate operations are very similar, we omit them here. Instead we want to introduce the rule for creating TargetLoads given in Figure 13. Again, we retype the original operation to the corresponding target operation. Since Load is a memory operation, we also need to set the volatile attribute.

The instruction selection rules are applied by the sequence shown in Figure 14. The [rule] operator matches and rewrites all occurrences of the given rule. Hence, we first create all immediate operations, and then the non-immediate operations for the remaining nodes.

## 4 Conclusion

We presented a GRGEN solution for both tasks, constant folding as well as instruction selection, covering all test cases. The solution employing pure graph rewriting is pretty concise for the constant folding task but admittedly less so for the instruction selection task, which requires a good deal of (very simple) graph relabeling rules, one rule per operation plus a further rule for the operations with immediate. The conciseness could be improved by stepping an abstraction level up by implementing a generator emitting GrGen code, which is what we did in [4] and [6] and what we recommend for the classes of applications which would lead to repetitive code.

The debugger included in the GrShell, which allows to execute the sequences step-by-step under user

control, was a great help during development of the optimizations. Especially its graph visualizations, which are depicted in Figure 3 and in the case description [2].

Due to the wavefront algorithm only visiting relevant operations and the core speed of the generated code we were able to give a solution with excellent performance. For the largest GReTL test case – consisting of 27993 nodes and 55981 edges – the execution speed is 6.3 seconds for constant folding and 111 milliseconds for instruction selection. For the largest test shipped with this case – consisting of 277529 nodes and 824154 edges – we need 11 seconds for constant folding and 1.2 seconds for instruction selection. This is more than one order of magnitude better than the next-closest competitor. The measurements were made using Ubuntu 11.04 and Mono 2.6.7 on a Core2Duo E6550 with 2.33 GHz. The numbers are even better for the SHARE machine(s) hosting the GrGen image [5] at the time of writing.

Our solution does not apply rules in parallel. However, we consider it as a benchmark for parallel solutions. Unfortunately, nobody submitted a parallel solution, which was the main purpose of publishing the challenge: we were hoping for hints about the potential benefits of parallelization. So the question which speed up can be gained by parallel rule application is still open.

## References

- [1] Jakob Blomer, Rubino Geiß & Edgar Jakumeit (2011): *The GrGen.NET User Manual*. <http://www.grgen.net>.
- [2] Sebastian Buchwald & Edgar Jakumeit (2011): *Compiler Optimization: A Case for the Transformation Tool Contest*. In Van Gorp et al. [7].
- [3] Sebastian Buchwald & Edgar Jakumeit (2011): *Saying Hello World with GrGen.NET – A Solution to the TTC 2011 Instructive Case*. In Van Gorp et al. [7].
- [4] Sebastian Buchwald & Andreas Zwinkau (2010): *Instruction Selection by Graph Transformation*. In: *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems, CASES '10*, ACM, New York, NY, USA, pp. 31–40, doi:10.1145/1878921.1878926.
- [5] Edgar Jakumeit & Sebastian Buchwald (2011): *SHARE demo related to the paper Solving the TTC 2011 Compiler Optimization Case with GrGen.NET*. [http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe\\_TTC11\\_GrGen\\_v2.vdi](http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_GrGen_v2.vdi).
- [6] Andreas Schösser & Rubino Geiß (2008): *Graph Rewriting for Hardware Dependent Program Optimizations*. In A. Schürr, M. Nagl & A. Zündorf, editors: *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (ACTIVE '07)*, LNCS, Springer, doi:10.1007/978-3-540-89020-1\_17.
- [7] Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors (2011): *TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011, Post-Proceedings*. EPTCS.

## A Code Listings

```

rule foldBinary(b:Binary<FirmNode>) : (Const)
{
  startBlock:StartBlock;
  b -df0:Dataflow-> c0:Const;
  b -df1:Dataflow-> c1:Const;
  hom(c0, c1);
  if{ df0.position==0 && df1.position==1; }

  alternative {
    foldAdd {
      if{ typeof(b)==Add; }

      modify {
        eval { c.value = c0.value + c1.value; }
      }
    }

    foldCmp {
      if{ typeof(b)==Cmp; }

      modify {
        eval {
          c.value = (
            cmp.relation == Relation::TRUE ||
            cmp.relation == Relation::GREATER
              && c0.value > c1.value ||
            cmp.relation == Relation::EQUAL
              && c0.value == c1.value ||
            cmp.relation == Relation::GREATER_EQUAL
              && c0.value >= c1.value ||
            cmp.relation == Relation::LESS
              && c0.value < c1.value ||
            cmp.relation == Relation::NOT_EQUAL
              && c0.value != c1.value ||
            cmp.relation == Relation::LESS_EQUAL
              && c0.value <= c1.value
          ) ? 1 : 0;
        }
      }
    }
  }

  ...
}

modify {
  c:Const -startBlockEdge:Dataflow-> startBlock;
  eval { startBlockEdge.position = -1; }
  exec([relinkUser(b,c)] ;> deleteNode(b));
  return(c);
}
}

```

Figure 4: Rule for folding Binary nodes.

```

rule foldPhi(phi:Phi<FirmNode>) : (Const)
{
  startBlock:StartBlock;
  phi -blockEdge:Dataflow-> block:Block;
  phi -e0:Dataflow-> c0:Const;

  iterated {
    phi -e:Dataflow-> c0;

    modify {
      delete(e);
    }
  }

  iterated {
    phi -e:Dataflow-> phi;

    modify {
      delete(e);
    }
  }

  negative {
    c0;
    block;
    phi -:Dataflow-> :FirmNode;
  }

  modify {
    delete(blockEdge, e0);

    c:Const<phi>;
    c -startBlockEdge:Dataflow-> startBlock;

    eval {
      c.value = c0.value;
      startBlockEdge.position = -1;
    }

    return(c);
  }
}

```

Figure 5: Rule for folding Phi nodes.



```

rule foldCond
{
  startBlock:StartBlock;
  cond:Cond -blockEdge:Dataflow->;
  cond -df0:Dataflow-> c0:Const;
  falseBlock:Block -falseEdge:False-> cond;
  trueBlock:Block -trueEdge:True-> cond;
  hom(falseBlock, trueBlock);

  alternative {
    TrueCond {
      if { c0.value == 1; }

      modify {
        delete(falseEdge);
        -jmpEdge:Controlflow<trueEdge>->;
      }
    }
    FalseCond {
      if { c0.value == 0; }

      modify {
        delete(trueEdge);
        -jmpEdge:Controlflow<falseEdge>->;
      }
    }
  }

  modify {
    delete(df0);
    jmp:Jump<cond>;
  }
}

```

Figure 6: Rule for folding Cond nodes.

```

rule removeUnreachableBlock
{
  block:Block\StartBlock;

  negative {
    block -cfgEdge:Controlflow->;
  }

  modify {
    delete(block);
  }
}

```

Figure 7: Rule for removing unreachable Blocks.

```

rule removeUnreachableNode
{
  n:FirmNode\Block;

  negative {
    n -blockEdge:Dataflow-> :Block;

    if { blockEdge.position == -1; }
  }

  modify {
    delete(n);
  }
}

```

Figure 8: Rule for removing nodes without a Block.

```

rule removeUnreachablePhiOperand
{
  phi:Phi -blockEdge:Dataflow-> block:Block;
  phi -operandEdge:FirmEdge-> operand:FirmNode;

  negative {
    block -cfgEdge:Controlflow->;

    if { cfgEdge.position == operandEdge.position; }
  }

  modify {
    delete(operandEdge);
  }
}

```

Figure 9: Rule for removing nodes without a Block.

```

rule foldAssociativeAndCommutative
{
  bin0:Binary;
  bin0 -:Dataflow-> block:Block;
  bin0 -binEdge:Dataflow-> bin1:typeof(bin0);
  bin0 -constEdge:Dataflow-> c0:Const;
  bin1 -:Dataflow-> operand:FirmNode\Block;
  bin1 -:Dataflow-> c1:Const;
  hom(c0, c1);

  if { bin0.associative && bin0.commutative; }

  modify {
    delete(binEdge, constEdge);

    bin:typeof(bin0);
    bin -blockEdge:Dataflow-> block;
    bin -left:Dataflow-> c0;
    bin -right:Dataflow-> c1;
    bin0 -binLeft:Dataflow-> operand;
    bin0 -binRight:Dataflow-> bin;

    eval {
      blockEdge.position = -1;
      left.position = 0;
      right.position = 1;
      binLeft.position = binEdge.position;
      binRight.position = constEdge.position;
    }
  }
}

```

Figure 10: Rule for constant folding of associative and commutative operations.

```

rule NormalizeConsts
{
  bin:Binary;
  bin -op0Edge:Dataflow-> op0:Const;
  bin -op1Edge:Dataflow-> op1:FirmNode\Const;

  if { bin.commutative && op0Edge.position == 0 && op1Edge.position == 1; }

  modify {
    eval {
      op0Edge.position = 1;
      op1Edge.position = 0;
    }
  }
}

```

Figure 11: Rule for normalizing commutative operations with a Const operand.

```

rule ToTargetAddI
{
  a:Add -df:Dataflow-> c:Const;
  if { df.position==1; }

  modify {
    ta:TargetAddI<a>;
    delete(df);
    eval {
      ta.value = c.value;
    }
  }
}

```

Figure 12: Rule for an TargetAddI operation.

```

rule ToTargetLoad
{
  l:Load;
  modify {
    t:TargetLoad<l>;
    eval { t.volatile = l.volatile; }
  }
}

```

Figure 13: Rule for an TargetLoad operation.

```

xgrs [NormalizeConst]
xgrs [ToTargetLoadI] | [ToTargetStoreI] | [ToTargetAddI] | \
    [ToTargetSubI] | [ToTargetMulI] | [ToTargetDivI] | \
    [ToTargetModI] | [ToTargetAndI] | [ToTargetOrI] | \
    [ToTargetEorI] | [ToTargetShlI] | [ToTargetShrI] | \
    [ToTargetShrsI] | [ToTargetCmpI]

xgrs [ToTargetJump] | [ToTargetCond] | [ToTargetLoad] | \
    [ToTargetStore] | [ToTargetConst] | [ToTargetSymConst] | \
    [ToTargetNot] | [ToTargetAdd] | [ToTargetSub] | \
    [ToTargetMul] | [ToTargetDiv] | [ToTargetMod] | \
    [ToTargetAnd] | [ToTargetOr] | [ToTargetEor] | \
    [ToTargetShl] | [ToTargetShr] | [ToTargetShrs] | \
    [ToTargetCmp]

```

Figure 14: Extended graph rewrite sequence for instruction selection.